Composable Distributed Mobile Applications and Services in Opportunistic Networks

Chrysa Papadaki
Technical University of Munich
papadaki.chr@gmail.com

Teemu Kärkkäinen
Technical University of Munich
kaerkkae@in.tum.de

Jörg Ott
Technical University of Munich
ott@in.tum.de

***Abstract*—Advances in computational devices, wireless networking, cyber-physical integration and novel user interfaces are creating a world where we are continuously surrounded by a wealth of computational resources and new ways of interacting with them. However, we currently lack means of composing applications and services that can take full advantage of this environment. In this paper we present a system design that allows application developers to treat the future environment as a generic execution environment, which will automatically distribute and execute the components of their applications. This has the potential to unlock a wealth of currently unused resources and enable new classes of more immersive and useful applications and service that execute directly in the surrounding environment.**

I. Introduction

The environment in which many future applications and services execute will be fundamentally different from today's execution environments. Since the earliest computing systems, the execution environment has taken two primary forms: 1) personal devices of the end-users, and 2) large, typically distributed, shared back-end systems. Applications either execute on a personal device and serve its user, or in a back-end and serve a large number of users. Modern networked applications often rely on both—a front-end executing on, e.g., a smartphone (as a separate app or within a web browser), and a back-end executing in datacenter connected by the Internet.

The primary technological trend of the recent years, however, has been the accumulation of more devices and computational capabilities directly within the users' immediate environment—phones, tablets, laptops; "smart" homes, offices and cities. Further, advances in local wireless communication capabilities potentially allow all these devices and infrastructures to communicate directly with each other. As these trends continue, the future execution environment looks neither like a singular personal device, nor a centrally controlled and managed datacenter. Rather the environment will appear as if the user was moving inside a datacenter filled with a wealth of different resources—all serving and interacting with the user.

While we have learned how to build applications and services for the two classic execution environments—personal computing devices and large datacenters—building applications that execute primarily in the users' immediate environment remains less well understood. Approaches taken by system providers to solve this complexity have largely been to build tightly integrated, closed, siloed solutions designed around a remote cloud service as the integration point (e.g, Apple HomeKit, Google Nest). The complexity for building an open, extensible system arises from trying to orchestrate a large number of heterogeneous devices with various types of connectivity options, owned and operated by different entities, to produce a reliable, coherent service to highly mobile users.

In this paper we tackle the above problem by designing and implementing a framework for building mobile applications out of composable components executing opportunistically in the devices surrounding the user. Our inspiration comes in part from the container management automation systems, such as Google's Borg, Omega and Kubernetes. These allow heterogeneous applications to be developed for—and their execution dynamically managed on—a datacenter infrastructure made up of thousands of machines with no explicit knowledge of the details of the execution environment. This is our goal as well, except we target the execution environment surrounding the user instead of a datacenter back-end infrastructure—in essence fog automation instead of cloud automation.

II. Future Execution Environment

We believe four technological trends will define the future execution environment for applications and services: 1) increase in local *computational capacity* in diverse devices in the environment, 2) increase in *wireless communication capacity* directly between co-located devices and from the devices to the Internet, 3) increasing capability of computing systems to interact with the physical world via sensors and actuators, and 4) new and immersive ways for interfacing users with the computing systems.

Moore's law has brought us increasingly powerful and increasingly ubiquitous computing devices starting from the 60s' mainframes, to the 70s' minicomputers, to the 80s' personal computers, to the 90s' laptops, to the 2000s' smartphones and tablets, and to the 2010s' smartwatches. The latest entrants in this continuum are cheap embedded computers such as the Raspberry Pi, and microcontrollers such as the ESP32, have allowed a much wider audience of enterprising hackers and makers to cheaply build imaginative computer driven systems. Embedded devices have resulted in an invisible accumulation of computational capacity in the environment around us: the MagSafe charger of a MacBook can perform three times as

978-1-5386-4725-7/18/$31.00

many calculations per seconds as the original Macintosh[1]; a Wi-Fi enabled SD card for a camera runs a full Linux server; and modern cars packed with embedded sensors and processors, even in gas-pedals. Projecting this into the future, it appears inevitable that in a few decades the environment within an arm's reach of us will have as much computational power as a rack of servers in today's data centers.

So far, the invisible accumulation of computational capacity in the environment has been driven largely by microcontrollers embedded in special purpose applications—either standalone or connected into systems via special-purpose communication buses (e.g., the CAN bus used in cars). New (and old) wireless technologies are enabling these devices to communicate with the world around them—e.g., via short range Bluetooth Low Energy, medium range Wi-Fi or long range via carrier-based and Ultra Narrow Band technologies. Advances in radio-frequency wireless communications are being joined by new spectrum ranges with different propagation characteristics—e.g., Visible Light Communications in the hundreds of THz range, mmWave technologies in the tens of GHz range, and White Space technologies scavenging capacity from legacy technologies in multiple spectrum ranges. Advances in these technologies will enable cheap, power efficient, and high capacity short and long range communications between the computational units that will fill our environment.

The increasing processing and communication capabilities of digital systems at lower cost and complexity thresholds are enabling another major trend: the increase in digital services' ability to sense and interact with the physical world. These technologies have been developed in multiple contexts and under many brands - Smart Cities, Smart Homes, Internet of Things, Industry 4.0 - but fundamentally they all combine environmental sensors and actuators connected to a centralized service via some network (typically the Internet). So far these systems have generally been special purpose, closed and tightly vertically integrated, but both Apple (HomeKit) and Google (Android) have made efforts to enable multiple third party devices to function in a single system - under a centrally controlled ecosystem. We foresee this trend resulting in most physical systems around us being capable of interacting with external computing systems and services.

Despite the great advances in the various underlying technologies, the fundamental purpose of computing systems—serving human needs—will remain the same. Thus new human interface technologies are often crucial in unlocking the full potential of computational advances—e.g., the way touch screens transformed cellular phones. It is interesting to note that at this stage of distributed computational evolution, no widespread innovations in human interface technologies has accompanied the explosion in the availability of embedded devices. Instead, these devices tend to feed into earlier-generation centralized, Internet-based services. We postulate that in the future the true capabilities of the advancing technologies will be unlocked by advances in human interface technologies that allow direct interactions between humans and their nearby computational environment. In the short to medium term we see augmented and virtual reality interfaces to be the main technologies of interest, while in the longer term it is possible these are supplanted, e.g., by direct brain-computer interfaces such as Elon Musk's Neuralink.

Combined, continued advances in the above four areas will result in a fundamentally transformed environment in which applications and services execute. It is unlikely that the current approaches to designing and building networked services are capable of taking full advantage of the new capabilities. Still, the major initiatives of both the industry and academia to build the foundation for the future systems are based on projecting the current system designs and architectures into the future, rather than projecting the evolution of the existing enabling technologies and latent human needs into the future and reasoning backwards what the system design should look like. For instance, the IoT vision is to connect every-thing into the Internet and make them accessible via centralized cloud based services. The 5G vision is to build fundamentally the same type of mobile networks as previously but with an order of magnitude higher bitrates, lower latencies, more devices and lower operational costs via virtualization techniques. The model of services as a centralized brain with centralized control operating in a remote data center with appendages reaching out to the users via the Internet and mobile operators is deeply ingrained in these designs. We seek to deviate from this conventional vision of the future and to put the human users at the center of the system architecture, allowing them to reach out to and directly leverage all the capabilities of the surrounding environment without going through the cloud or centralized service providers.

III. Conceptual Design

The basic approach that we follow consists of three parts: 1) *Decomposing* the service into its constituent components of computation. 2) Dynamically *instantiating* a set of components, that comprise an entire service, in the environment. 3) *Wiring* the I/O of the components together—i.e., setting up the control flow. In this section we define the key concepts of *composable component*, out of which the high level services are composed, and *event-based interactions* between the sets of distributed components. We show our concrete framework design to enable these concepts in practice in the Section IV, and show how to apply them to build a distributed application in Section V.

A. Composable Components

The first part of our approach, as stated above, is to decompose the complex service into smaller, self-contained units of functionality, which can dynamically distributed over multiple devices in the execution environment surrounding the user. The key concept of our design is the *composable component*—a simple, self-contained unit of functionality, which can be composed with other such units at runtime. This

[1]MagSafe has an MSP430 (16 MHz, 4.6 MIPS), the original Macintosh has a 68000 (7.8 MHz, 1.4 MIPS).

facilitates the componentization for the design and development of distributed applications and services, i.e., the creation of a suite of components that collaborate to provide the desired functionality. This reduces the complexity of designing services that span multiple physical devices in a continually changing environment, since the designer does not need to understand and design for the specific environment where the service will execute—it is enough to design the decomposition (components and their relationships) of the service.

Composable components must have the following characteristics: **Small**: small manageable piece of work that provides a distinct, single-purpose functionality. **Modular**: self-contained with well-defined APIs to interact with other components. **Opportunistic**: interacts opportunistically with other components, which reside in different processes on the same or a remote device. The duration of the presence of the component in the system at runtime is unspecified. **Independently deployeable**: comes as code executable by its own runtime or by a generic framework. **Indepentently pluggable** and **replaceable**: can be (un)plugged to the system and operate without any manual administration, can be replaced and upgraded without affecting the other components. **stateful** or **stateless**: can be stateful by encapsulating a data store or stateless, in which case it interacts with a remote database to either transform further the stored content and act as a service or display content to the end-user.

We define two types of components: *services* and *widgets*. Service components can be either stateful or stateless, and contain business logic useful for other components. A service component might implement a particular computational task, e.g., running an image recognition algorithm on a provided image, or it might provide a storage service, e.g., a database queryable via SQL-statements received as events or through RPC calls. Widget components, in contrast, are used for interacting with the (human) users. They are stateless and present a user interface to the end-user, e.g., by displaying an interactive interface on a large touch display (e.g., smart kiosk) to some data from a service component. The system might define multiple widgets for the same service, designed for different user interface devices, ranging from LEDs and buzzers, to smart watches and smartphones, to smart kiosks and projection displays, to futuristic virtual and augmented reality devices. In classical terms, the widgets would correspond to the frontend and service components the backend of the system. We provide a more detailed description of the components in Section IV.

B. Event-Based Component Interactions

The second part of service creation, after the decomposition into self-contained functional units, is to facilitate the run-time interactions between the components. Our design divides this problem into two distinct contexts—*inside* and *outside* of a device. This division is important, because the environment within a device is largely static and centrally controlled, while the environment around the device is dynamic, ever-changing and under no central authority. Which in turn means that inside a device we can tightly bind sets of components together, while any interactions with components outside the devices are necessarily opportunistic in nature.

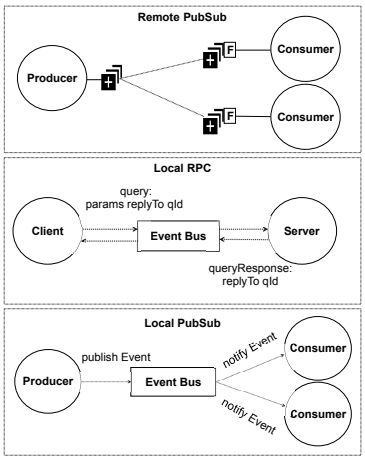

Fig. 1. Supported interaction models for the service components.

We take an *event bus* based approach—popular in, e.g., modern backend systems—which allows interacting components to be decoupled and freely added and removed at runtime. Conceptually, the components attach to a shared bus and interact by sending and receiving events. The events are (topic, data)-tuples, and the components *publish* them into the bus and *subscribe* to receive events for specific topics—the topic names can be flat or have an explicitly designed information structure. I.e., components publish events for content dissemination, database operations or other requests, and interested components take actions based on those events. We follow this topic-based publish-subscribe messaging paradigm for component interactions both inside and outside the devices.

More specifically, as shown in Figure 1, we provide: 1) external topic-based publish-subscribe events, 2) internal remote procedure calls (RPC) between components, and 3) internal topic-based publish-subscribe events.

For device-internal interactions, we rely on a non-persistent event bus that routes events from publishers to subscribers in real-time—e.g., a message broker that routes messages from producers to consumers as shown at the bottom of the figure. The local RPC mechanism can be overlaid on the publish-subscribe model by extending the event to a (remote-procedure, reply-id, query)-tuple, where the *remote-procedure* identifies the procedure to be called, *reply-id* is used to associate responses to queries, and the *query* contains any parameters and other details required by the procedure. The mapping from the RPC event tuple to the ordinary event tuple can be done transparently by the framework (see, e.g., [9]).

Device-external interactions require a different abstraction because the surrounding environment is dynamic and uncontrolled. We take the approach of *opportunistic networking* [22], which we abstract as a persistent *remote event* queue. Local components will publish and subscribe to the remote event queue similarly to the local event bus, but the delivery semantics are different. In particular, the devices will opportunisti-

cally synchronize their remote event queues when they form communication contacts between each other, meaning that there are no guarantees about delivery, delays, or ordering of the remote events. Multiple opportunistic networking systems and implementations exist that fit this abstraction[24], [15], [8], so we do not define a new one.

C. Runtime Service Composition

The final part of the service creation, after decomposition into a set of components and their interaction relationships, is to arrange for them to be executed in the environment. For this we rely on the same opportunistic networking mechanisms as for the remote interactions between components described above. Each component of a service is compiled into executable form and combined with a set of metadata describing it to the composition system. In particular, the metadata describes the resource requirements and dependencies of the component, and any security parameters (e.g., the publisher's signature). Then each bundle of a component, data and metadata is published into the remote event queue (i.e., the underlying opportunistic networking platform) and disseminated into various devices in the environment. The execution system in these devices loads and executes the components, subject to the restrictions in the metadata. We describe the full framework that manages the runtime service composition next in Section IV.

D. Security

Given that the system is designed to execute arbitrary computations received from the environment, and to disseminate the input and output data opportunistically between devices, security mechanisms are needed to guard against attacks and improper use of the resources. While detailed security mechanism design is outside the scope of this paper, we describe the main security issues arising from both the opportunistic computation execution and communications, and propose approaches to address them.

The first set of security issues arise from the execution of arbitrary computations received from the network. An attacker could craft computation that quickly consumes unlimited amounts of memory and processor time, denying service to other computations—and this can be amplified exponentially if the computations are allowed to create new computations (similar to a "fork bomb"). To prevent this, the system needs a runtime that limits the amount of resources any component can consume both instantaneously and over time. Beyond just consuming excess resources, attack code may attempt access unauthorized resources via exploits or due to a lack of access control mechanism. This could include, for example, accessing private services of the underlying operating system or interfering with the execution of other components in the system. The runtime can employ isolation and sandboxing mechanism (e.g., the *secure computing mode* of the Linux kernel) to limit the calls that the code can make. The components could be digitally signed by the developers or deployers, and the runtime can choose to execute code only from trusted

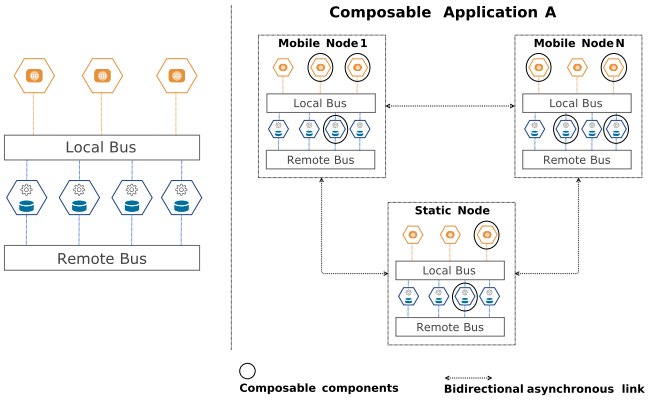

Fig. 2. Composable System Architecture

sources—or at least have the ability to trace misbehaving code back to its origin. Beyond the infrastructure protecting against malicious code, the user may also wish to ensure that the environment is executing the provided code without modifications, so that the resulting service can be trusted. For this, trusted computing hardware support mechanisms could be employed as a basis for a solution.

The second set of security issues arises from the movement of data between the computations. As the data can transit multiple devices, as with all opportunistic networking, the content will be visible to third parties. To maintain privacy the content of the messages should be encrypted. This, however, poses a problem since the components must be able to decrypt the data in order to operate on it, but the components themselves are transmitted opportunistically—also exposing any included key material third parties. This means that a general solution for data privacy would need to be based on homomorphic encryption mechanisms. Similarly to computational resource use attacks, an attacker could seek to disrupt the communications by flooding the network messages. To combat this, the amount of messages generated by computations could be limited and filtering mechanisms employed to message exchanges.

IV. Framework Design

This section presents the design of a framework for enabling the composable design concepts described in the previous section. We detail the elements of the framework and their interactions, and the design rationale behind them. We do not describe the implementation here due to space constraints, but the details can be found in [20] and the source code in [1].

Figure 2 shows how the high level concepts of service and widget components, and local and remove event buses are structured within the framework. The left part of the figure shows a single node instance with the high level elements. At the bottom is the *remote event queue*, which will loosely synchronize events with other framework instances over opportunistic communication opportunities. Connected to the remote event queue is the set of *service components*.

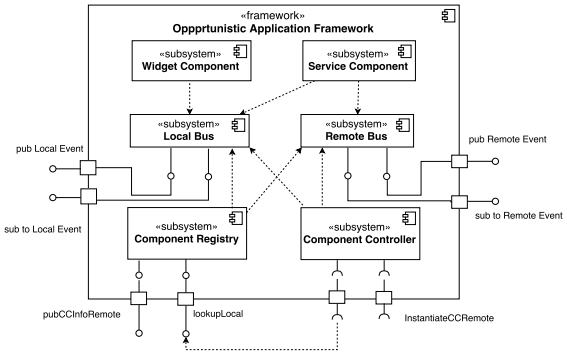

Fig. 3. UML component diagram of the opportunistic application framework.

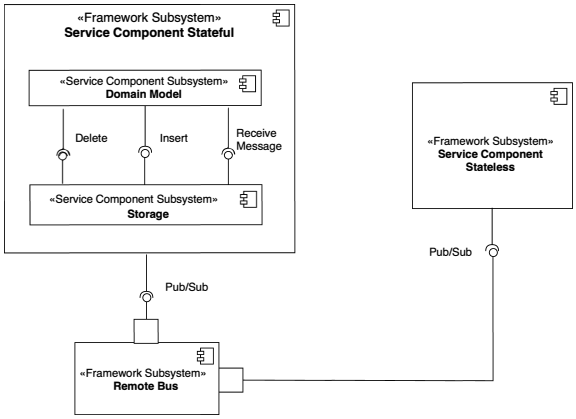

Fig. 4. UML component diagram of stateful and stateless service components

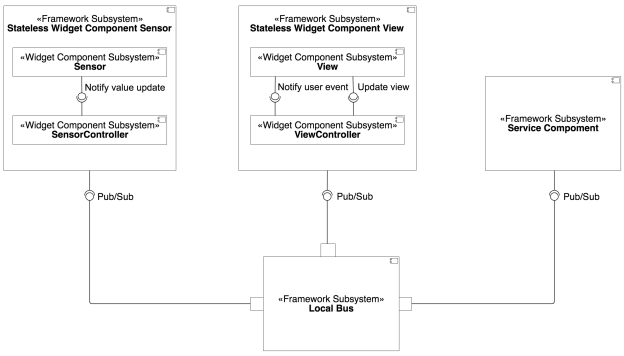

Fig. 5. UML component diagram of widget components.

These will communicate with remote component instances via the remote event queue, but also provide services to local components via the *local event bus*. The user-interface for the local user is provided via the *widget* components at the top of the figure, that exclusively use the service provided by the local service components—widgets interacting with remote service components would lead to poor user experience due to the unpredictable communication characteristics of the underlying opportunistic networking. A full service instantiation will require a number of components running on a (dynamic) set of devices, as shown on the right of Figure 2. The figure shows an example of a composable application built out of four service components and five widgets (marked with a black circle) executing on three different node instances. We give a concrete example of such an application decomposition and instantiation in Section V.

More detailed view of the framework architecture is given by Figure 3, which shows the internal subsystems of a framework instance. Beyond the buses and components described earlier, the framework needs elements for managing the components—the *component registry* and the *component controller*. The component registry maintains information on the components installed within a node instance while the component controller uses the it to check the state of the running components, manage their lifecycles and install new components. We describe the details of each element in the following sections.

A. Service Component

Service components are the units of composition used for business logic and (persistent) state. As described earlier, the framework provides the service components interfaces for internal and external interactions via the eventing system, which the components use to both consume and provide services to other components. In addition, they may have arbitrary interfaces towards the platform capabilities, such as the filesystem or a locally running database instance—these are implementation specific details not covered here. Each service component is packaged independently as self-contained binaries—along with its dependencies and metadata, as described earlier—and distributed inside remote events between component controllers described later in this section.

Figure 4 show an example design of a stateful and stateless service components connected via a remote event bus. The stateful variant shown is divided into a (persistent or in-memory) *storage layer* that contains the state, and the *domain model* layer that contains the business logic. In this configuration, the storage layer acts as a cache to the business logic, storing and passing on incoming events as they are received from remote components. As shown on the right of the figure, the service components can also be stateless, simply mapping incoming events to outgoing ones without maintaining any state. A typical pattern is to have both stateful and stateless components interacting, for example, by installing stateful data aggregation components in stationary nodes in the environment and running a stateless component on a mobile device that picks up remote events from the stateful components and passes them onto interface widgets.

B. Widget Component

Widget components represents a platform-dependent user application that contains all the views and view controllers to display the content to the user and handle the user interactions. The only framework dependency they have is the local event bus interface, through which they read and

write user-generated content. The widgets are not aware of the underlying networking infrastructure and only consume and produce local events (publish-subscribe and RPC). They can be full standard (mobile) user applications with platform dependent interfaces, or individual components displayed as a part of, e.g., a dashboard-style combined interface, or even just provide minimal user interaction via, e.g., LEDs and buttons.

Figure 5 shows an example with the framework local bus, a sensor and a view widget component. Both view and sensor widgets consume the event interface provided by the local bus in order to publish and subscribe to local events. The view widget is composed of the *View* subsystem which contains all the application view elements and the *ViewController* subsystem which is responsible for receiving the local events published by the local bus and handle them properly in order to pass the required information to the views. In addition, it receives the user input and publishes it to the local bus. In the case of the sensor widget, it is composed of the *Sensor* subsystem which is a platform sensor module that provides an interface for getting notified when a new sensor value has been recorded, and the *SensorController* subsystem which gets the sensor value and publishes it to the local bus so that the service components can also get the value—the interaction could also work the other way, with a service component reading the sensor value and publishing the value as an event to widgets.

C. Remote Bus

The *remote bus* abstraction is the fundamental architectural element that enables components distributed in the environment to directly interact with each other by agreeing on a common set of remote events. It provides remote publish-subscribe messaging as described in section III-B by encapsulating an opportunistic networking system.

While our prototype implementation is based on the Liberouter neighborhood networking system[14], any system that provides the following features can be used (e.g, [24], [8]): **Peer discovery**: Detecting peers that can be reached via various communication technologies either directly (Wi-Fi Direct, Bluetooth, LTE D2D) or via an infrastructure network (Wi-Fi, LTE), and opening communication contacts to them. **Routing and dissemination**: Deciding which messages should be passed to which peers. Could be, e.g., Delay-Tolerant Networking routing algorithms or opportunistic data dissemination algorithms. **Storage**: The remote bus is not a real-time abstraction, instead the remote events are assumed to have a significant lifetime, during which they are replicated between framework instances—and to enable this, the events must be stored for an extended period of time. This matches the store-carry-forward paradigm typically employed by opportunistic networking. **Publish-subscribe messaging**: The abstraction used is a topic-based publish-subscribe, rather than end-point centric send-receive messaging.

D. Local Bus

The *local bus* is an abstraction of real-time, non-persistent event distribution between local components—typically implemented in-memory, possibly within the same address space as the components (in a monolithic framework implementation). It is used to decouple the widgets layer from the services layer—the presentation from the business logic—and to realize the local publish-subscribe and RPC interactions between components. The widgets layer is only connected to the local bus since they are used for user interaction, which requires immediate interactions and tight control loops between components, which is not possible over the remote bus described in the previous section. If we want a widget that listens to a remote service component, we need to deploy a local service component that interacts with the remote service component via the remote bus and publishes events to the widget via the local bus.

The local bus approach enforces separation of concerns and the single responsibility principle, which leads to a more modular design that implies increase of composability. Further, since the components are not tightly coupled, they can be individually added and removed at runtime without affecting others. This enables upgrading, expanding, load balancing and scaling of the application or service at runtime simply by deploying new components.

E. Component Registry

The Component Registry is responsible for keeping track of the components, and persisting the component binaries and metadata. It runs on every node and exposes an interface via the remote bus to access information about the components such as name, state (active, running, disabled), type and version. This enables the discovery of components in the network and can be used in complex scenarios where, e.g., an application is composed out of specific components and there are multiple in the network with similar functionality. An event message of type registry query can be published to the topic *component-registry*, requesting of the list of running components, and based on this information the proper service binding can be decided. A concrete example is a user-composable application where the users can choose what service and widget components to bind together, e.g., the user decides to use the *university-party-photos* service component and bind it to the *awesome-gallery-view* widget instead of the *simple-gallery-view* widget. In addition, it provides a local interface via the local bus to allow the component controller to register and manage the components, as described next.

F. Component Controller

The *component controller* manages the components on a node. It runs independently and handles the lifecycle of a component—it is able to install, uninstall, stop, start and replace components. It provides an interface for other components in the network to publish their services and widgets, including their metadata. Then it decides whether the received component should be installed on the node or not—applying security, resource management and other policies. It queries the component registry through the local bus to look up, register and unregister components.

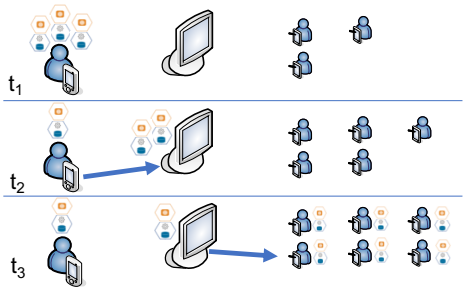

Fig. 6. Opportunistic classroom polling application scenario.

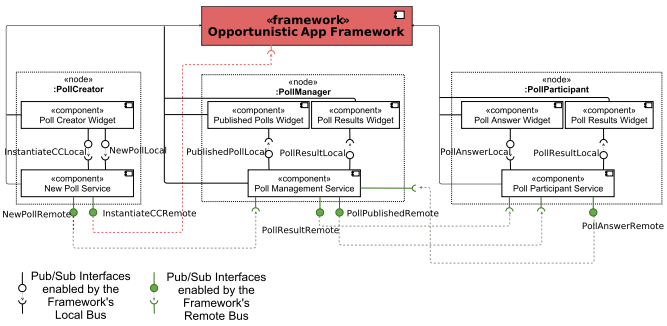

Fig. 7. UML component diagram of the polling application

V. Designing Composable Applications

This section presents an example design of an opportunistic classroom polling application to demonstrate the application of the framework in practice, and to be used for the evaluation in Section VI. Due to space constraints we do not present the details of the implementation here, but they can be found in [20]—along with two further application examples—and the source code can be found in [1].

Figure 6 shows the high level application scenario of a lecture in a classroom attended by a teacher and a number of students, all of whom we assume to be carrying smartphones running our framework. Further, the classroom is assumed to hold fixed infrastructure, in particular a large projection screen driven by a computer also running our framework. We want to create a polling application that allows the teacher to create a poll question, which will be displayed on the projection screen, and allow the students to answer the poll from their smartphones. The app should calculate the results and display them on the projection screen and the teacher's device. Importantly, we do not want anything to be pre-installed in the classroom or student devices, but rather dynamically instantiate the service in the various devices for the duration of the class, and have it disappear after the class is over. This process is shown in the figure, where at time t_1 the teacher enters the classroom carrying all the components of the polling service. At time t_2 the framework will disseminate the components to the classroom infrastructure (computer attached to the projection screen) and from there to the students' devices at time t_3 as they enter the classroom. Once the components have been deployed, they provide the different user interfaces and business logic in the various devices to carry out the polling process—and get removed from the devices as the students and the teacher leave the classroom.

Figure 7 presents the system decomposition using a component UML diagram. We define three different roles for the devices: 1) *PollCreator*, which allows the user to create and publish new polls. It is also the service creator and publishes the components to the other nodes. 2) *PollManager*, stores and manages the polls and aggregates the poll answers to generate the results. 3) *PollParticipant*, which offers a UI to the users to respond to the polls and display the results.

PollCreator comprises two components, the Poll Creator Widget (PCW) and the New Poll Service (NPS). PCW provides a user interface to the users for creating and publishing polls. It also stores the binaries for all the other components, which it will disseminate into other devices in the surrounding environment. PCW publishes both the binaries and the new polls as local events to the local bus. The business logic for the creator is contained in the NPS. It receives local events *NewPollLocal* for receiving a new poll and *InstantiateCCLocal* for receiving a component binary and metadata from the PCW. As soon as it receives the local events, it stores the *NewPollLocal* in its local database and prepares the remote events, *NewPollRemote* and *InstantiateCCRemote*, containing the user content and component binaries respectively, and publishes them to the remote bus.

The PollManager node is designed to run in a static infrastructure node, serving the same role as backend server would in a classical design. It is loaded by the framework from a remote *InstantiateCCRemote* event published by the PollCreator. It is comprised of widget components, the *Published Polls Widget* (PPW) and the *Poll Results Widget* (PRW), and a service component, *Poll Management Service* (PMS). The PMS is maintains the published polls and calculates the poll results. As soon as it receives a *NewPollRemote* event, it publishes a remote *PollPublishedRemote* event to notify the interested remote components about the published poll.

The PollParticipant node provides a user application for responding to the published polls. It is composed of a *Poll Participant Service* (PPS), the *Poll Answer Widget* (PAW) and the *Poll Results Widget* (PRW). The PPS receives new polls via the *PollPublishedRemote* events and publish them locally to the PAW, which displays the poll to the user. When the user responds to a poll, the PAW creates a *PollAnswerLocal* event and publishes it to the local bus. The PPS receives the user answer, stores it and publishes a *PollAnswerRemote* to the remote bus.

The PMS component on the PollManager node receives the user answer and updates the poll results based on the new value. Subsequently, it publishes the *PollResultRemote* and *PollResultLocal* events so that the interested local and remote PRW compoennts receive the new poll result. The PPW and PRW display the published polls and the poll results respectively on the projection screen so that everyone can view the content. Additionally, the users' mobile devices which act

as PollParticipant nodes get the content on their screens, as does the teacher's PollCreator node.

VI. Evaluation

We evaluate a prototype implementation[2] of the framework and the polling application in a small testbed with mobile and stationary devices. The evaluation focuses on two cases: 1) static, minimal topology to show the characteristics of the service instantiation and operation, and 2) dynamic scenario that shows how a service can be composed in a dynamically changing environment.

A. Static Scenario

The static evaluation scenario is comprised of two mobile devices (Android 4.4.2, 1.4/1.6 GHz A7/A9, 2/1.5 GB RAM), which act as the PollCreator and PollParticipant, and a laptop (Windows 8.1, 1.7 GHz i5, 8 GB RAM) which acts as a PollManager (see Section V)—we call these *Client A*, *Client B* and *Liberouter* respectively. The devices are connected to the same Wi-Fi access point for the entire duration of the test. Each device starts each experiment with the framework software installed, while Client A carries all the components of the poll application. We first analyze the *instantiation phase* where the components are distributed and initialized in the devices, and then the *functional phase* where the poll service is operational. In the functional phase we automatically create polls every 30 seconds via the PCW from Client A and the Client B automatically responds to the received polls. We collect logs generated by the framework, components and the Scampi middleware, and compose a timeline and delays of events in the system from them.

1) Service Instantiation Phase: Figure 8 shows the observed flow of remote event messages that lead to the instantiation and configuration of the poll service. Specifically, at time t_0, Client A (PollCreator) publishes the event *ConfigMessage* which includes the PollManagementService.jar and configuration metadata for instantiation by the receiving node. The configuration information includes the node role—*SERVICE_INFRASTRUCTURE*—and thus it is handled and instantiated by the Liberouter node (1.1 Instantiate PMS.jar). In the meantime, at time t_1, Client A (PollCreator) publishes another *ConfigMessage* with the PollParticipantService.jar with the role *SERVICE_MOBILE*, which is handled by Client B (2.1 Instantiate PPS.jar) acting as the PollParticipant node. Finally at time t_3, Client A publishes the last *ConfigMessage* for the PollParticipantWidget component. The role of this component is *WIDGET_MOBILE* and is handled and instantiated by nodes of type *PollParticipant*. Each of the timestamps t_2, t_4 and t_5 denote the completion of the component instantiation on the corresponding nodes.

We measured the mean latencies and 95% confidence intervals (CI) for the three instantiation steps ($t_{PMS} = t_2 - t_0$, $t_{PPS} = t_4 - t_1$, $t_{PPW} = t_5 - t_3$). For the t_{PMS}, that represents the instantiation of the PMS component of size

[2]Implementation description is omitted here due to space constraints—it can be found in [20]. The source code is available in [1]

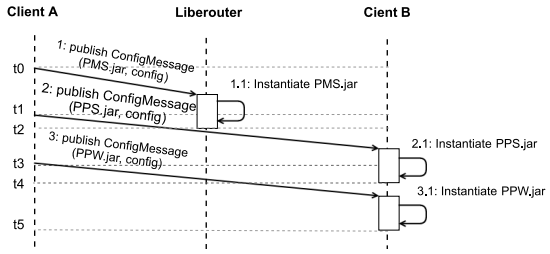

Fig. 8. UML sequence diagram of instantiation phase.

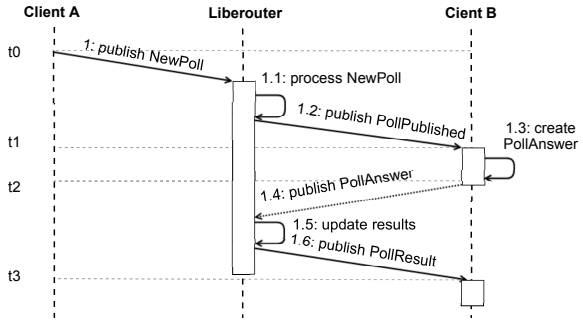

Fig. 9. UML sequence diagram of functional phase.

24 KB in the Liberouter node, the mean delay 14.6 s with 95% CI [8.0 s, 21.2 s]. The delays for instantiating the PPS, t_{PPS} are similar, with a mean of 11.2 s and 95% CI [6.2 s, 16.2 s]. The delay to instantiate PPW, t_{PPW}, is even longer—average of 54.1 s and 95% CI [37.5 s, 70.6 s]—as this involved manual user interaction to gain the permission to install the widget component that arrives as an Android APK file. Since the transmission times in our setup are low, the relatively long delays reflect the overheads of dynamic code loading and execution from JARs—we believe a production quality implementation would reduce these by an order of magnitude.

2) Functional Phase: The flow of events between the devices recorded during the operation of the application—creating, answering, aggregating and displaying poll results—is shown in Figure 9. The interactions consists of local and remote events, but here we present only the remote interactions due to space restrictions—further analysis of local interactions are reported in [20].

More specifically, Client A publishes the remote event *NewPoll* at time t_0, which contains the question and answer options for a new poll. After the event arrives on the Liberouter, the PollManager extracts the new poll information, checks if the poll already exists in the database and if not, stores it locally and publishes an external event *PollPublished*. At time t_1 the *PollPublished* arrives at Client B, which processes the event and automatically selects the first poll answer option—the delay from the poll creation at t_0 to the participant receiving it at t_1 was on average 1.50 s with 95% CI [1.49 s, 1.51 s]. At time t_2, the participant publishes the *PollAnswer* event

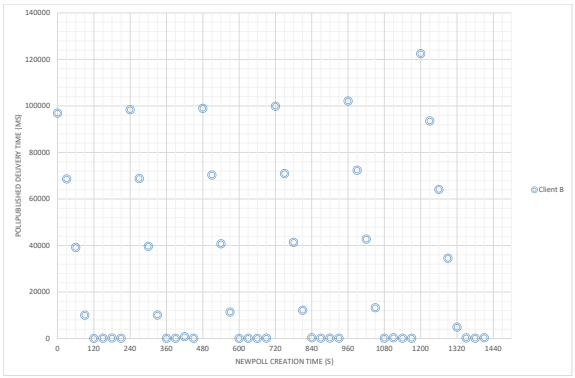

Fig. 10. PollPublished remote event delivery delay in ms to Client B.

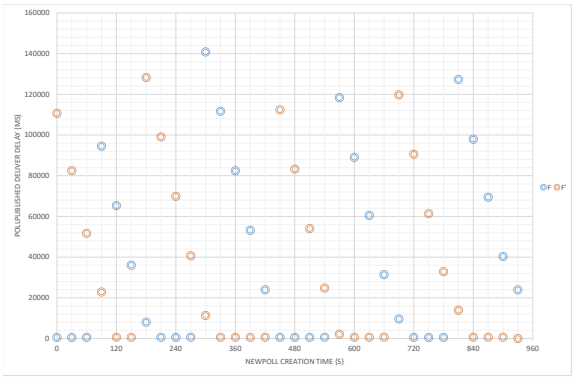

Fig. 11. NewPoll remote event delivery delay in milliseconds on Liberouter in function of NewPoll local event creation time in seconds on Client A

to the remote bus. The full delay from the poll event being received from the network, displayed by the user widget, to the answer being published into the network was on average 0.78 s with 95% CI [0.75 s, 0.81 s]—this represents all the interactions between the components in the local system When the PollAnswer arrives at Liberouter, the PollService component updates the poll results and publishes a new *PollResult* event to the remote bus. At time t_3, on average 0.61 s (95% CI [0.60 s, 0.62 s]) after publishing the answer, Client B receives the *PollResult*. This shows that once the system has been fully initialized, the interactions between the participating nodes are fast enough that the user perceives instantaneous operation even though the service is composed of components running on three separate devices.

B. Dynamic Scenario

Since we target a highly dynamic execution environment, we also need to study the behavior of the composed poll application in the face of changes in the environment. We do this by looking at two different changes: 1) *Temporary node absence*, where a node temporarily leaves the area where the poll application is deployed and then returns. 2) *Infrastructure failure*, where the static node running the PollManager service fails and another instance takes its place. In both cases the system should automatically recover and rebuild the application state. We use the same testbed and scenario setup as in the static functional phase test described in the previous section, but we introduce scheduled failures to the Client B Wi-Fi connectivity and the Liberouter framework instance.

1) Temporary Node Absence: In this experiment, we demonstrate the capability of the PollParticipant node to reconstruct the current and previously published polls state after a disconnection from the network. The Wi-Fi on Client B is being switched on and off every 120 seconds, resulting in equally long connected and disconnected periods, while Client A publishes new polls every 30 seconds.

Figure 10 illustrates the end-to-end delays for the *NewPoll* and *PollPublished* local and remote event messages from the perspective of Client B as a function of the creation time of the local event *NewPoll* by the PCW component on Client A. A sawtooth pattern with a period of 240 seconds (twice the switching period) is formed due to the remote events being buffered in the remote queue while Client B is disconnected. The delay reaches its peak just after the end of the of each connected period and approaches its minimum values during the connected period—just as expected for a delay-tolerant message delivery. The maximum delay shown in Figure 10 for the first part of the functional phase is approximately 122 seconds, which corresponds to the disconnected periods due to Wi-Fi switching off. During the connected periods the delays are close to zero which leads to real-time experience that resembles centralized infrastructure network behavior. These results show that the participants—in this case students—can enter and leave the area of application deployment freely and the system will automatically synchronize the application state.

2) Switching Framework Instance: With this experiment, we demonstrate the capability of the system to recover from failures of the central PollManager node. In this case we run two framework instances, F and F', in the central (Liberouter) node. Every 120 seconds we stop one of the instances and start the other one, which simulates the failure of one device and appearance of another. We seek to prove that when one device in the environment fails, another one can automatically take over its duties. Similarly to the previous test cases, Client A and Client B act as PollCreator and PollParticipant respectively and Liberouter as PollManager.

Figure 11 shows the delays of the remote events, due to the polls created every 30 seconds by Client A, delivered to the Liberouter. We observe the same sawtooth pattern with 240 second period as in the previous case, again caused by buffering of events by the remote event bus while the node is not reachable (in this case due to being switched off)—although here we observe two of these patterns corresponding to the two different framework instances F and F'. We can further observe that despite the framework instances going on and off, there is always an active one being able to process the event at close to zero delay—in other words, the system is seamlessly handing over the tasks from the failed node to the new active one. These results show that the central PollManager role can be dynamically moved between devices in a dynamic environment.

VII. Related Work

Our work relates to multiple different areas of research within the space of pervasive computing[23], opportunistic networking[22], opportunistic computing[6] and cyber-physical systems[5]. The idea of allowing mobile clients to exploit the resource in the environment traces back to the late 90s, with seminal work on location-aware services[12], [2]. We aim to carry this vision to the future environments, where the clients are not seeking and using services already existing in the environment, but rather using the environment to execute their own custom applications. The idea of composing services by exploiting resource of nearby devices via opportunistic contacts is also not new—a theoretical model and analysis given by Passarella et.al[21], special purpose designs have been done for specific use cases such as sensor inference[11] and middlewares[4], [19] have been proposed to help developers with various aspects of mobile computing. Our design goes beyond this state of the art, providing a simple, practical platform for service composition and execution, that meets the demands of the future computational environment.

Under the umbrella term *mobile edge computing*[18], [25], significant work has been done on systems that execute parts of mobile users' applications on generic computational component in the near environment or further in the network. A rough division can be made between *computational/code offloading*[17], [10] where a primarily user device resident application pushes a resource heavy computation to the cloud or a nearby device, and operator driven mechanisms, such as *fog computing*[3] and *5G edge computing*[13], where a primarily network resident service pushes latency sensitive computation to the edge of the network near the user. In mobile code offloading, the focus of research has been on improving performance[16] and saving energy[7] by making offloading decisions—our focus is using the environment to dynamically compose entire services from many interacting computations that could not be run on the users' device alone.

VIII. Conclusion

In this work we presented our vision of what the future computational environment will look like and what opportunities it provides for application and service developers. We believe our framework for decomposing services into components and opportunistically distributing then in the environment will enable future developers to take full advantage of all these opportunities. In this paper we laid the groundwork by presenting the system design, showing how it can be used to create a non-trivial service that executes on multiple devices, and proved the feasibility of it through a prototype evaluation. There does, however, still remain a large number of open issues that will need to be tackled—security, resource management, intelligent component orchestration, and supporting very low power platform, to name a few. We are planning to study all these issues in future work, as well as build larger testbeds and more applications to evaluate the practicality of the system.

Acknowledgments

This work was supported by the European Commission Horizon 2020 Programme RIFE Project Grant No. 644663.

References

[1] Microframe source code. https://goo.gl/7HXB23.
[2] P. Bahl and V. N. Padmanabhan. Radar: An in-building rf-based user location and tracking system. In *IEEE INFOCOM*, 2000.
[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.
[4] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on software engineering*, 2003.
[5] M. Conti, S. K. Das, C. Bisdikian, M. Kumar, L. M. Ni, A. Passarella, G. Roussos, G. Tröster, G. Tsudik, and F. Zambonelli. Looking ahead in pervasive computing: Challenges and opportunities in the era of cyber–physical convergence. *Pervasive and Mobile Computing*, 2012.
[6] M. Conti, S. Giordano, M. May, and A. Passarella. From opportunistic networks to opportunistic computing. *IEEE COMMAG*, 2010.
[7] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.
[8] C. Dannewitz. Netinf: An information-centric design for the future internet. In *GI/ITG KuVS Workshop on The Future Internet*, 2009.
[9] M. J. Demmer, K. R. Fall, T. Koponen, and S. Shenker. Towards a modern communications api. In *HotNets*. Citeseer, 2007.
[10] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya. Mobile code offloading: from concept to practice and beyond. *IEEE Communications Magazine*, 53(3):80–88, 2015.
[11] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *ACM MobiCom*, 2016.
[12] T. D. Hodes, R. H. Katz, E. Servan-Schreiber, and L. Rowe. Composable ad-hoc mobile services for universal interaction. In *ACM MobiCom*, 1997.
[13] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. Mobile edge computingâĂŤa key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.
[14] T. Kärkkäinen and J. Ott. Towards autonomous neighborhood networking. *IEEE WONS*, 2014.
[15] T. Kärkkäinen, M. Pitkänen, P. Houghton, and J. Ott. Scampi application platform. In *ACM MobiCom CHANTS*, 2012.
[16] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.
[17] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, 2013.
[18] P. Mach and Z. Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.
[19] C. Mascolo, L. Capra, and W. Emmerich. Mobile computing middleware. In *NETWORKING*. Springer, 2002.
[20] C. Papadaki. Master Thesis: An Architecture for Composable Distributed Applications and Services in Disconnected Information-centric Networks. https://goo.gl/VpcJgp, 2016.
[21] A. Passarella, M. Kumar, M. Conti, and E. Borgia. Minimum-delay service provisioning in opportunistic networks. *IEEE Transactions on Parallel and Distributed Systems*, 2011.
[22] L. Pelusi, A. Passarella, and M. Conti. Opportunistic networking: data forwarding in disconnected mobile ad hoc networks. *IEEE COMMAG*, 2006.
[23] D. Saha and A. Mukherjee. Pervasive computing: a paradigm for the 21st century. *Computer*, 2003.
[24] S. Schildt, J. Morgenroth, W.-B. Pöttner, and L. Wolf. Ibr-dtn: A lightweight, modular and highly portable bundle protocol implementation. *Electronic Communications of the EASST*, 37, 2011.
[25] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang. A survey on mobile edge networks: Convergence of computing, caching and communications. *IEEE Access*, 5:6757–6779, 2017.